Debug Messages in EUROPA

Overview

In EUROPA, all debug messages are categorized using *markers*. This allows developers to:

- Selectively print messages by editing the Debug.cfg file.
- Selectively manage messages within C++.

A *marker* can be any string with no leading or following whitespace and no comment characters. Such *marker*s are the first argument to the debugMsg macro used to create debug messages. For example:

```
debugMsg("Schema:isA", "Checking if " << descendant.toString() << " is a " << ancestor.toString());
```

If messages with the 'Schema:isA' *marker* are enabled, output like the following will be produced:

```
[Schema:isA]Checking if AgentTimeline.Go is a AgentTimeline.Go
```

Debug information is by default redirected to std::cout; however, when running executables with `jam`, output is generally redirected to a file RUN_*execname*

A: Configuring With Debug.cfg

Debug.cfg can be easily configured to define which debug *markers* will be enabled.

Syntax

In Debug.cfg, *markers* are specified by a line composed of an optional ':' followed by any text which isn't a comment or trailing whitespace. Any *markers* that start with that text are enabled (ie. any messages with those *markers* will be printed).

Debug.cfg supports three different comment characters '#', ';', and '/'. These characters and any text which follows them is ignored.

Debug.cfg ignores whitespace when:

- prior to the first non-whitespace character on a line
- after the last non-whitespace character on a line which precedes a comment.
- both of the above conditions are met (a line with no marker, this is completely ignored)

An example

```
# A comment
; Also a comment
/ Our final comment we could have used // but it's not necessary
 :ConstraintEngine   # our first debugMsg marker enabling messages in the ConstraintEngine.

# For a more robust example, examine the Debug.cfg generated by makeproject.
```

B: Managing Debug Messages Within C++

To enable debug messages in code, the stream to write them to must be assigned. For example, to use cerr:

```
DebugMessage::setStream(std::cerr);
```

All messages can then be enabled with:

```
DebugMessage::enableAll();
```

An individual one can be enabled with:

```
DebugMessage *msg;
msg->enable();
```

An individual debug message can be looked up using:

```
msg = DebugMessage::findMsg("file", "marker");
```

If this matches more then one existing debug message, the first one found will be returned. To find all messages in a given file, e.g.:

```
std::list<DebugMessage*> msgs; DebugMessage::findMatchingMsgs("file", "", &msgs);
```

where the second argument is a empty (zero length) std::string. Note that msgs is not cleared (emptied) by this function, only added to. An empty string can also be passed for the file name, so:

```
DebugMessage::findMatchingMsgs("", "", &msgs);
```

will have the same effect as:

```
msgs = DebugMessage::getAllMsgs();
```

except the latter is (currently) a const reference to the internal list and thus runs much faster but cannot be modified.

In all cases, individual messages will not appear in such lists unless the code in question (where the debugMsg() call appears) has already been executed; until then, the info about the individual debug message simply isn't available. Removing this restriction would require the complete list of debug messages to be constructed at compile time (similar to how the entire list of parameter constraint functions is presently done at compile time). However, the calls:

```
DebugMessage::enableAll();
```

```
DebugMessage::disableAll();
DebugMessage::readConfigFile(istream& is);
```

are not restricted to existing messages, as they store the "patterns" that are presently enabled and, when a new message is created that matches any of the enabled patterns, it is immediately enabled (and therefore prints its message immediately after being created, as part of the debugMsg() macro). The method:

```
DebugMessage::enableMatchingMessages("file", "marker");
```

adds the appropriate pattern to this internal list of enabled patterns, which is checked immediately for existing debug messages and also when a new debug message is created.

There is no corresponding disableMatchingMessages() in the current implementation, but that could be very tricky (or costly at run time) to implement for cases like:

```
DebugMessage::enableAll();
DebugMessage::disableMatchingMsgs("", "marker");
DebugMessage::disableMatchingMsgs("file", "");
DebugMessage::enableMatchingMsgs("", "marker");
```

since there is no explicit list of files or markers mentioned in debug messages.